Matching in Multi-Agent Pathfinding using M*

Jonathan Dönszelmann[1] , **Jesse Mulderij**[1] , **Mathijs de Weerdt**[1]

[1]TU Delft
jdonszelmann@student.tudelft.nl
{j.mulderij, m.m.deweerdt}@tudelft.nl

Abstract

Multi-agent pathfinding (*MAPF*) can be extended by assigning agents to teams. In a team, agents need to be assigned (or matched) to a team goal such that the *sum of individual costs* is minimised. This extension is called *MAPFM*. *M** is a complete and optimal algorithm to solve *MAPF*. In this paper, a modification is proposed to *M** to allow it to solve *MAPFM* problems as well. To accomplish this, two strategies are proposed, called *inmatching* and *prematching*. In this paper, it is shown that *prematching* is generally preferable to *inmatching*, the benefits of different optimisations for *M** are compared, and it is shown how well *M** stacks up against other *MAPF* solvers extended to perform matching.

1. Introduction

A large number of real-world situations require the planning of collision-free routes for multiple agents [1]. For example, the routing of trains over a rail network [1], directing robots in warehouses [2] or making sure autonomous cars do not collide on the road [3]. Problems of this nature are called *Multi-agent pathfinding* problems, which will hereafter be abbreviated to *MAPF*.

In *MAPF*, each agent has a starting position and a goal position. For every agent, a route needs to be found from their start to their goal, without two agents colliding. Finding these collision-free routes has been proven to be *NP-hard* [4].

One algorithm to solve *MAPF* is called *M** [5]. *M** is derived from the *A** algorithm as described by Standley [6]. *A** generally plans the paths of agents together. This means that in each timestep, the number of possible next states grows exponentially with the number of agents. Contrasting that, in *M**, agents follow an individually optimal path whenever possible. In each timestep, only the subset of colliding agents is jointly planned.

The *MAPF* problem can be generalised by grouping agents into teams. A team consists of one or more agents. In contrast to *MAPF*, an agent does not have a single goal in this generalisation. Instead, agents can use any goal associated with the team it is in, but no two agents can end at the same goal. An algorithm solving this problem must assign each agent to a goal. Such an assignment is called a matching. This problem is therefore named multi-agent pathfinding with matching (hereafter abbreviated to *MAPFM*).

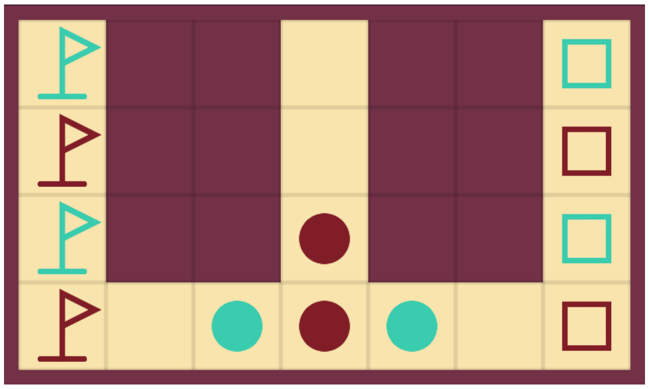


Figure 1: An snapshot of agents solving a *MAPFM*. Two teams of agents are moving from their starts to their goals. Squares, flags and circles are starts, goals and agents respectively. One red agent (at 4 squares from the left, 3 squares from the top) waits to let the three other agents pass.

In this paper, *MAPFM* will first be defined, after which two techniques will be discussed which extend *M** to also solve *MAPFM* problems. These two techniques will be compared, both to each other, and to a number of other algorithms which also solve *MAPFM* problems. Apart from this comparison, a number of extensions to *M** will be introduced which improve *M**'s perfor-

[1]An agent is the collective name for a single unit for which a path is calculated. The word ‘agent’ could refer to a single robot, a train or a car, but can also refer to an abstract or simulated entity. In MAPF, paths are planned for multiple agents.

mance. Some of these extensions can also improve the performance of *M** on plain *MAPF* problems.

2. Prior work

Before this research, a separate study has been performed [7] on a problem called the target-assignment and pathfinding (*TAPF*) problem. The difference between *TAPF* and *MAPFM* is that in *TAPF*, the *makespan* (the cost of the longest path) instead of the *sum of individual costs* (the cost of all paths combined) is minimised. To solve *TAPF*, [7] uses conflict based search (*CBS*) [8], using a min-cost flow based algorithm for the lower level. The min-cost flow is used to solve *TAPF* for a single team in polynomial time. The higher level of *CBS* combines single team solutions to find a solution for all teams. In parallel research to this paper, [9] shows that such a min-cost flow based algorithm can also be used to solve *MAPFM* problems.

To the best of the author's knowledge, solving *TAPF* or *MAPFM* with other algorithms than *CBS* (such as *M**) has not yet been explored.

In contrast, a lot of research has been done finding and improving algorithms which solve *MAPF* [5, 6, 8, 10–13]. The current state-of-the-art *MAPF* algorithms are *Conflict Based Search* (*CBS*) and *Branch-and-Cut-and-Price* (*BCP*). However, which of the existing algorithms for *MAPF* can be adapted such that they also perform well at solving *MAPFM* problems is yet unclear.

3. Definitions of *MAPF* and *MAPFM*

The definition of multi-agent pathfinding used in this paper is based on Stern's definition given in [14]. The definition is given below.

A *MAPF* problem P consists of the following elements:

$$P = \langle G, s, g \rangle$$

- G is an undirected graph $\langle V, E \rangle$
  - V is a set of vertices
  - E is a set of unweighted edges between vertices
- s is a list of k vertices where every s_i is a starting position for an agent a_i
- g is a list of k vertices where every g_i is a goal position for an agent a_i

The goal of *MAPF* is to find a path for all agents a_i from s_i to g_i, without vertex and edge conflicts. This means that in a timestep t, two agents may not be on the same vertex, and between two timesteps, two agents may not travel over the same edge.

The cost c_i of a path π_i is the number of timesteps until the last time that agent a_i arrives at its goal g_i. [14] defines two primary approaches to define the cost c of a solution to a *MAPF* instance. The *sum of individual costs* is the sum of the cost of all paths ($c = \sum_{n=0}^{k} c_n$). Alternatively, the *makespan* is equal to the maximum cost of all paths ($c = \max_{n=0}^{k} c_n$). An optimal solution to a *MAPF* instance is a set of paths π with the smallest possible cost c.

Although the algorithms presented in this paper would work on any graph G, in examples and experiments, G is simplified to a 4-connected grid. In this paper, this grid is sometimes called a 'map', on which agents move from their start positions to their goal positions.

MAPFM

Now, the concept of matching is added to the definition of *MAPF*, to create multi-agent pathfinding with matching (*MAPFM*). The new *MAPFM* problem adds two elements to P, as shown below:

$$P' = \langle G, s, g, sc, gc \rangle$$

- *sc* is a list of colours. Each starting vertex s_i is assigned a colour sc_i.
- *gc* is a list of colours. Each starting vertex s_i is assigned a colour sc_i.

This definition divides all agents into teams. An agent a_i's team colour is the colour of its starting location sc_i. In total there are K teams $k_1 \dots k_n$. K is equal to the number of different colours in *sc* and *gc*.

In *MAPFM*, agents do not have a single goal. Instead, teams have several goals all with the same colour gc_i. A team has as many goals as there are agents. A solution to *MAPFM* is a set of paths for each agent, in which every agent travels to a goal with a colour equal to the colour of that agent.

MAPFM uses the *sum of individual costs* as an optimisation criterion.

4. A description of *M**

*M** [5] is a complete and optimal pathfinding algorithm, similar to *A**. However, *M** is specifically designed for *MAPF*. In *M**, the search space consists of the combination of positions of all the agents on a grid. To use *A** for *MAPF*, the planning of paths for all agents needs to be coupled to make sure no states are expanded in which collisions occur. Coupled planning means that the states used in *A** contain the positions of all agents at the same time, and every expansion expands new states for all agents at the same time.

Unlike in *A**, in *M** the planning of agents is initially not coupled. Instead, *M** assumes that the optimal path for an agent to their goal does not contain any collisions with other agents. As long as this is true, the planning of agents is separated.

However, the assumption that all optimal paths are collision-free obviously does not always hold. Therefore, each state in *M** also holds, apart from the positions of each agent, two sets called the *collision set* and the *backpropagation set* as described in [5]. When *M** detects that a state contains collisions, the algorithm does not continue expanding this state. It instead uses information stored in these two sets to backtrack and find the shortest route around these collisions. Agents associated with the collision temporarily plan routes in a coupled fashion.

After collisions have been circumvented, *M** plans agents independently again according to their individually optimal path. It does, however, record information about the previous collisions (in these *backtracking-* and *collision sets*). This is to make sure that when in the future another collision occurs, it can either resolve this collision locally or backtrack back to the previous collision to circumvent this old collision differently, potentially avoiding the new collision altogether.

Just like *A**, *M** uses a priority queue and a heuristic to prioritise the exploration of states. This finds optimal solutions as long as an admissible heuristic is used such as the Manhattan- or euclidean distance to the closest goal location for each agent.

5. *M** AND MATCHING

To add matching to *M**, this paper proposes two options which are called "inmatching" and "prematching". In this section, both are explained and their advantages and disadvantages are discussed.

5.1. INMATCHING

Inmatching is the process of performing matching as a part of the pathfinding algorithm that is used. To understand it, it is useful to first look at inmatching in *A**. With *A**, the expansion of a state consists of all possible moves for all agents. *A** searches through the search space, until the goal state has been removed from the frontier. With an admissible heuristic, *A** guarantees that backtracking from this first goal state gives a shortest path between the start and goal location.

With *inmatching*, there is not one goal state. Instead, any state in which all agents stand on a goal of their own colour is considered a goal state. This means there is more than one goal state. An admissible heuristic for *inmatching M** is the distance to the nearest goal state.

Inmatching applied to *M**

Inmatching can similarly be used with *M**. However, there is something that makes it much less efficient in *M** than in *A**, the implications of which will be evaluated in section 8.1.

One of the core ideas of *M** is that agents, when not colliding, follow an individually optimal path to their goal. With *inmatching*, every goal in the team of the agent is a valid goal. Therefore, to preserve optimality, agents should consider optimal paths to each of the goals in the team.

Before discussing the significance of this, a concept the *expansion size* needs to be explained first. In *A**, a priority queue is used storing states. Each time a state is removed from the queue it is expanded into a number of child states which go back into the queue. The number of child states which are created is the *expansion size*.

In regular *M**, when there are no collisions, the *expansion size* is 1. Each agent simply follows their individually optimal path to their goal. There is only one such optimal path so there is only one next state. Only when there are collisions, the *expansion size* is bigger than 1, which is less efficient. But with *inmatching*, there is not one optimal path. Due to the introduction of the teams, an agent can have multiple goal locations, which may all have a different individually optimal paths. Thus, the expansion size is often bigger than 1 even when there are no collisions. In fact, the expansion size grows exponentially with the number of goals in a team. The following equation gives an upper bound for the total expansion size of one state:

$$\prod_{n=1}^{K} \text{goals}(k_n)$$

Here, the *goals* function gives the number of goals in team k_n, and K is the number of teams.

*inmatching M** will hereafter be abbreviated to *imM**

5.2. PREMATCHING

```
Algorithm 1: prematch $M^*$
  Result: Find the matching with smallest cost
  ________________________________________
  $matchings \leftarrow$ find all matchings$(starts, goals)$
  foreach $m \in matchings$ do
  |  $S(i) \leftarrow$ mstar$(starts, m)$ {Evaluate with $M^*$}
  end
  return $min(S)$ {by calculated cost}
```

Alternatively, there is *prematching.* With *prematching* the *MAPFM* problem is transformed in a number of *MAPF* problems. Each possible matching is calculated in advance, and normal *M** as described by [5] is performed on each matching exhaustively. This exhaustive search is shown in algorithm 1.

In section 6.3 an extension to *prematching M** is proposed which uses a heuristic to prune some of the matchings which otherwise needed to be evaluated.

*prematching M** will hereafter be abbreviated to *pmM**

To summarize, *inmatching* treats multiple states as goal states, and searches until it reaches the first such goal state. In *M**, to guarantee optimality, agents should take individually optimal paths to all goal states of the team they are in. *Prematching* instead exhaustively evaluates *M** on every matching.

6. EXTENSIONS TO *M**

*M** can be improved upon in various ways. Some of these extensions are only applicable to *M** with matching, while others improve *M** itself.

In this section these extensions are described in detail.

6.1. Precomputed paths and heuristics

A part of solving *M** is finding the individually optimal path for each agent. This can be done with conventional pathfinding algorithms such as *A**. But to avoid repeated calculation, it is important to first create a lookup table of the minimal distance to each goal, for every open square on the map. With this table, finding in which way an agent should move to reach the goal in an individually optimal manner becomes a constant time operation.

When using *prematching*, the same goal locations and start locations are (although in a different permutation) reused multiple times. The lookup table can remain the same for all evaluations of matchings.

Using this lookup table, the heuristic of *M** can be improved as well. Instead of using the *euclidean-* or *Manhattan distance* to the goal, the lookup table can be used to find the exact distance. The heuristic found with this approach also takes the obstacles on the map into account.

6.2. Operator decomposition

In [6], Standley describes a technique called operator decomposition (*OD*). Operator decomposition is designed to be an extension to *A** to improve the runtime of solving of *MAPF* problems.

In *MAPF*, when an agent moves, it can perform one of five actions (assuming a grid map is used): wait on the same square, or move in one of the four cardinal directions. With *A**, the movement of all agents is coupled. When a state is expanded, every child state contains the new position of every agent. This means that if there are 5 agents, which may all perform one of 5 possible actions, $5^5 = 3125$ child states are created and added to the frontier. This is called a full expansion.

Operator decomposition can lower this large expansion size by introducing the concept of partial states. As the name implies, when a state is removed from the frontier, it is not fully expanded. Instead, only the 5 possible actions of a single agent are evaluated. These 5 partial states are added to the frontier. When a partial state is removed from the frontier, the actions of the next agent are evaluated too until the actions of all agents are evaluated and a full expansion is reached again.

So in the worst case, *OD* still performs a full expansion. However, the advantage of *OD* is that partial states can also be prioritised in the frontier. When the expansion of a subset of agents is found to result in a low heuristic, the partial state is given a high priority to be further expanded. This may lead to finding solutions to *MAPF* instances more quickly.

Even though *M** tries to avoid the coupled planning of agents as much as possible, when collisions occur, expansion sizes can still grow large. [15] showed that as a result *M** can also benefit from *OD*. In section 8.2 the benefits *OD* has on *M** when solving matching problems is evaluated.

6.3. Pruning of matchings

Algorithm 2: prematch *M** with pruning

```
Result: Find the matching with smallest cost
        without evaluating every matching.
$m_{best} \leftarrow \varnothing$

$M \leftarrow$ find all matchings($starts, goals$);
sort($M$) {on heuristic; ascending}
foreach $m \leftarrow M$ do
    if $heuristic(m) < cost(m_{best})$ then
        $s \leftarrow$ mstar($starts, m$);
        $m_{best} \leftarrow min(m_{best}, s)$;
    end
end
```

*pmM**, as previously described, has to evaluate every matching of every team. However, it is possible to discard some matchings without evaluating them with *M** at all, by using heuristics.

To do this, the heuristic (i.e. the sum of distances between start and goal locations) of the initial state of every matching is calculated. Because this heuristic is admissible, it represents a lower bound for the cost of this matching.

Then, when *prematching* evaluates every matching m, it keeps track of the best matching m_{best} which is the matching with best cost so far. However, when heuristic(m) $\geq$ $cost(m_{best})$, m can immediately be pruned, because this matching m can never yield a solution which has a lower cost than m_{best}.

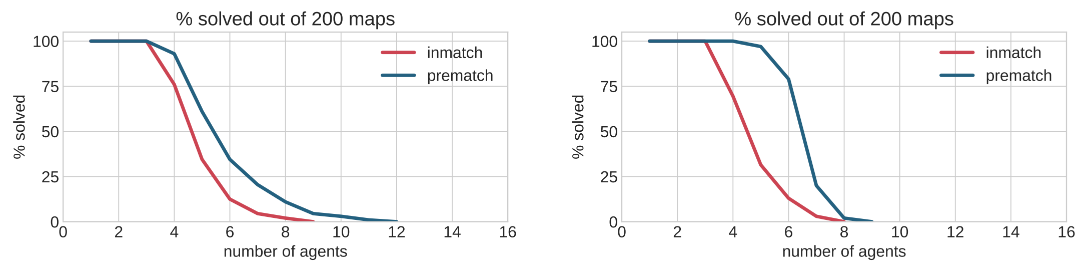
Sorting

To take maximum advantage of pruning, matchings can be sorted based on their heuristic. Making sure the matching with the lowest heuristic is evaluated first can increase chances that later matchings are pruned. Algorithm 2 shows how pruning with sorting works.

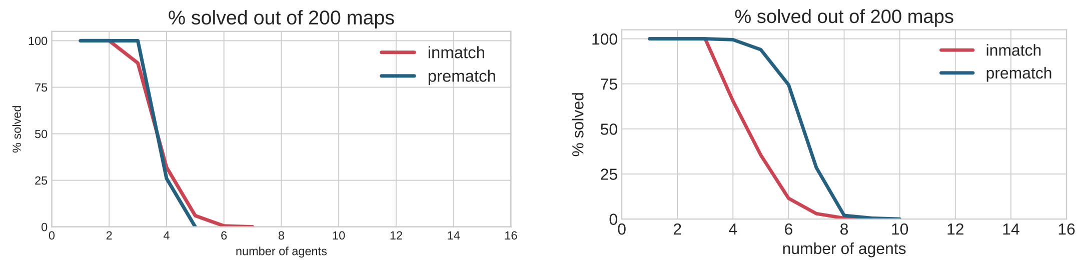
7. Experimental setup

To evaluate the performance of matching *M**, a number of experiments were performed. For each of these experiments, the algorithm used is written in Python 3.9 and benchmarks were run on a virtualized system with a 12 core Intel Xeon E5-2683 running at 2GHz, which has 8Gib of RAM.

There are various factors deciding how a *MAPF* solving algorithm performs in benchmarks. Three important factors are the total number of agents for which paths need to be found, the number of teams over which the agents are distributed, and the layout of the maps on which benchmarks are run.



(a) Agents split over 3 teams. Left: 75% wall, right: 25%wall



(b) All agents in 1 team. Left: 75% wall, right: 25%wall

Figure 2: Percentage of maps solved in 2 minutes out of 200 random 20×20 maps.

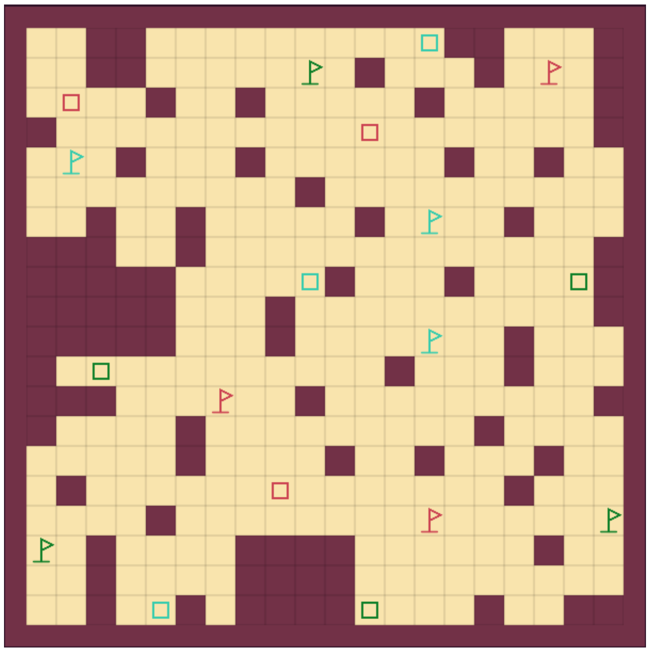


Figure 3: A map which is 25% obstacle, with agents distributed in three teams. This map was one of many used in the performed experiments.

The experiments are grouped into sets of four *scenarios*, to show the differences in performance as these parameters vary. Performance is assessed on maps where either 25% or 75% of the grid is an obstacle, and in situations where agents are grouped into either 1 or 3 teams.

For these benchmarks, 20×20 grid maps[2] are randomly generated using an algorithm derived from maze generation algorithms created by the author of [16]. Four sets of maps were generated for each experiment with the previously discussed parameters. Agents and goals are uniformly distributed on this map and teams are assigned randomly. When the team size does not divide the number of agents, there will be one smaller team. Maps are generated such they are always theoretically solvable. An example of such a randomly generated map can be found in figure 3.

Because *MAPFM* is an *NP-Hard* problem, it is possible that finding the solution to an instance of the problem takes an unreasonable amount of time. Therefore, in each experiment, a timeout of 2 minutes is used. Experimental results show the percentage of maps (out of a set of 200) which the algorithm manages to solve within this timeout.

8. Results and Discussion

8.1. Matching strategies

In section 5, two strategies were proposed for adding matching to $M*$. Both strategies were tested following the experimental setup described in section 7. In figure 2 the results of this experiment are shown.

From these graphs, it can be seen that *prematching* is generally superior to *inmatching* in all but one of the scenarios.

Expansion size

In section 5.1, it was hypothesised that this difference in performance could occur, because of the larger expansion size.

[2]Maps of different sizes will likely also show differences in performance. However, experiments with other map sizes are not included in this paper.

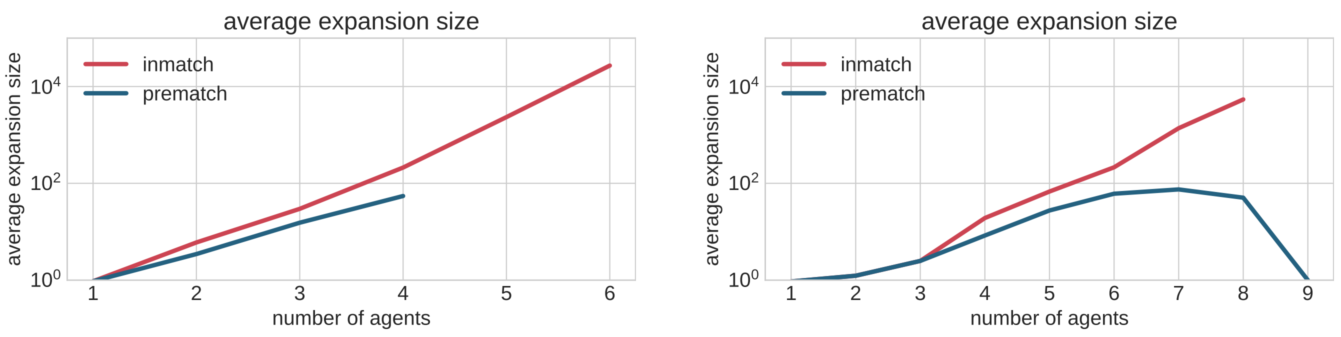


Figure 4: Average expansion size on the same maps as used in figure 2b (i.e. all agents in one team). Left: 75% wall, right: 25% wall. The blue line in the left figure stops earlier because *prematching* was not able to solve maps with more agents.

To demonstrate this is indeed the case, another, separate, experiment was performed. Again on 200 random maps as described in section 7, but this time showing the average number of states expanded each expansion.

In figure 4, on a logarithmic scale, the outcomes of this experiment are shown. Indeed, on average, inmatching expands many more states.

When *inmatching* is better

However, in one of the scenarios in figure 2b *inmatching* does seem to perform slightly better than *prematching*. In this benchmark, all agents are in a single team, and the map is 75% filled with obstacles.

When agents are all in one team, there are more matchings compared to when agents are separated into multiple teams. With *prematching*, each matching needs to be exhaustively searched, which is slow.

In contrast, the problem of *inmatching* expanding so many states is reduced in this scenario. There are fewer directions for agents to go because of the large number of obstacles.

So in general, when there are many matchings, but few options for agents to move in, *inmatching* can perform better.

8.2. Extensions to $M*$

Various extensions have been proposed in section 6. To show the benefits of each extension, they were compared following the experimental setup described in section 7.

In figure 5, the performance of these extensions is shown. Each line in the graph shows a new extension added to all of the previous extensions. For example the line displaying operator decomposition also uses precomputed heuristics and all other previous extensions.

Because in section 8.1, *prematching* was observed to generally perform better, only the *prematching* matching strategy is used in this experiment.

Pruning and sorting

In the graphs where 25% the maps is filled with obstacles, pruning and sorting have a large impact on performance. However, in graphs where 75% is a wall there is barely a difference at all. The heuristic used to prune, is the Manhattan distance between the agent start locations and the closest goal. This does not take into account the obstacles in the map.

Pruning is only possible when the heuristic is larger than the best matching found so far (as described in section 6.3). This does not happen often with this inaccurate heuristic. Therefore, using the precomputed heuristic (also shown in figure 5) makes a large difference.

When there are fewer obstacles in the map, the probability of pruning is much higher, explaining the difference in performance of pruning and sorting between these two scenarios.

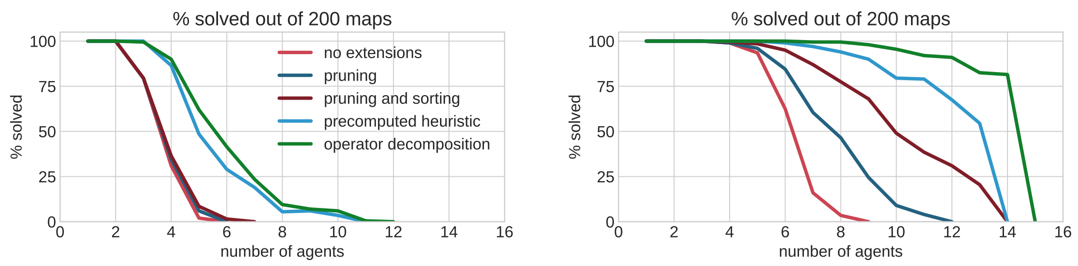
Memory usage

The best variant of $M*$ can be seen solving maps with 14 agents in figure 5a. After this, the percentage abruptly drops to 0. This is not caused by the complexity of the problems, but rather by the memory that $M*$ uses here. Attempts with maps with 15 agents saw memory usages of over 8GiB to solve a single instance, which caused attempts to be stopped by the operating system.

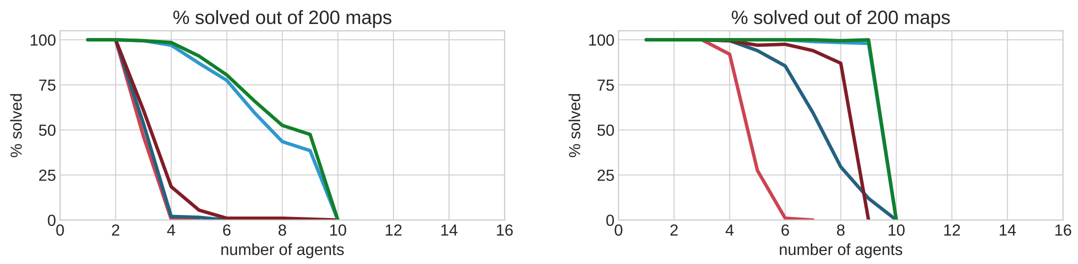
8.3. Comparison with other algorithms

This research is part of a set of parallel studies on how to extend a collection of *MAPF* algorithms with matching. All these studies used exactly the same problem definition but using one of the following base algorithms:

- Extended partial expansion $A*$ (*EPEA**) [11] (implementation and research by [17])
- $A*$ with operator decomposition and independence detection ($A*$+OD+ID) [6] (implementation and research by [16])
- Increasing cost tree search (ICTS) [13] (implementation and research by [18])
- Conflict-Based Min-Cost-Flow (CBM) [7] (implementation and research by [9])



(a) 3 teams. Left: 75% wall, right: 25%wall



(b) 1 team. Left: 75% wall, right: 25%wall

Figure 5: Percentage of maps solved in 2 minutes out of 200 random 20×20 maps. Each line and extension to *M** is graphed in combination with all previous extensions. For example, the line displaying *operator decomposition* also uses precomputed heuristics and all other previous extensions.

Experiments were performed following the experimental setup described in section 7. All algorithms were benchmarked on the same computer to ensure a fair comparison. Still, it is hard to compare algorithms well because, for example, *CBM* made use of external libraries written in *C++*. But, compared to the exponential nature of *MAPF*, such a strategy providing a linear speed-up is relatively insignificant.

The results of this comparison can be found in figure 6.

In figure 6b, it may look like an error was made. *CBM* solves 100% of the maps up to maps with 25 agents. In fact, *CBM* is able to solve maps with many more agents. *CBM* makes use of min-cost flow to solve *MAPFM* for single teams in polynomial time. Solutions for single teams are checked, and modified when conflicts are found between agents in different teams.

But, this means that in cases where there *is* only one team, *CBM* is able to solve *MAPFM* in polynomal time, scaling linearly with the number of agents for which a route needs to be found.

In the experiments where multiple teams were used (as shown in figure 6a), *CBM* performs more comparably with other algorithms. Still, when there are few obstacles, *CBM* excels.

Apart from *CBM*, all other algorithms (including *M**) show very similar performance characteristics. All these algorithms use a form of prematching, where all matchings are exhaustively searched. This seems to be a factor which limits the capabilities of all these algorithms.

Only *A*+OD+ID* performs exceptionally well on maps 25% filled with obstacles, and with 3 teams. The reason for this is likely due to the way independence detection works in the implementation described in [16].

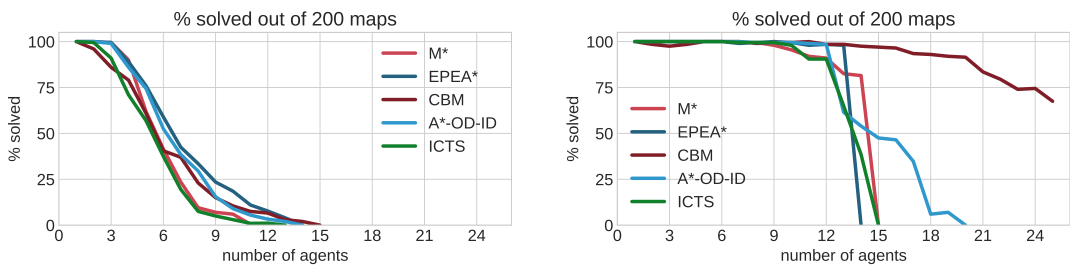
It must however be said, that this comparison is **not** complete. For example, testing with agents split over more teams, on larger maps, or with specifically crafted obstacles such as long corridors may show very different results.

9. Conclusion

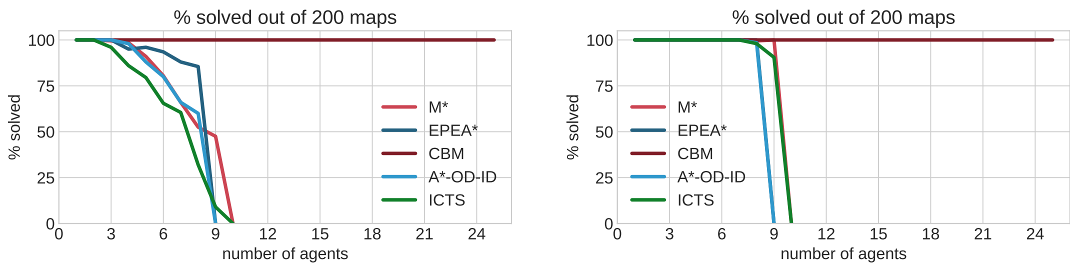
In this paper, a generalisation of multi agent pathfinding (*MAPF*) was introduced, called matching. This new problem, is called *MAPFM M**, an algorithm for *MAPF*, was modified to solve these *MAPFM* problems. To do this, two strategies were explored called *inmatching* and *prematchings*. Experimental results showed that in many cases, *prematching* is superior to *inmatching*.

Subsequently, several improvements to *prematching M** were considered and their benefits were experimentally evaluated. Pruning and sorting of matchings, using precomputed heuristics, and independence detection were shown to have a large effect on performance.

Finally, it was shown that *pmM** has very similar performance characteristics as *ICTS*, *EPEA** and *A*+OD+ID*. *CBM* performed notably better than *M** in experiments performed in this paper.



(a) 3 teams. Left: 75% wall, right: 25%wall



(b) 1 team. Left: 75% wall, right: 25%wall. In the right graph, the lines for *EPEA** and *A*+OD+ID* overlap, and so do the lines for *ICTS* and *M**.

Figure 6: Percentage of maps solved in 2 minutes out of 200 random 20×20 maps.

10. Future work

Partial Expansion

One problem with the *inmatching* strategy presented in section 5.1 is that the expansion size is very large. However, a lot of the expanded states are never needed or accessed again. Even with *prematching* this was a problem. For example, while creating the graphs in figure 5, the data points where 14 or 15 agents were involved could use as much as 8GiB of memory.

Partial- or *enhanced partial expansion* as presented in [11] and [12] only expands states when they are needed to reduce memory usage. It is likely that partial expansion can also be applied to *M** to improve its memory performance.

Conflict based search

In this paper, matching *M** was compared to several other algorithms. One of those was CBS [8]. However, this implementation used a min-cost flow based approach similar to the one described in [7], but altered to find the *sum of individual costs* instead of the *makespan*.

CBS, without matching and these min-cost flow extensions, is one of the most studied algorithms for *MAPF* ([8, 19–22]). Comparing *M** with *inmatching* and *prematching* to *CBS* using similar approaches to add matching to it, could provide useful data.

Waypoints

In previous research [23–26], *MAPF* was extended with waypoints instead of matching. It may be possible to combine these two extensions. For example, each team may have a set of waypoints. It does not matter which agent visits which waypoint as long as all waypoints are visited. Alternatively, each agent has their own waypoints, but the goals are still shared with a team of other agents.

Agents and goals

In the definition of matching presented in this paper, in each team the number of goals was always equal to the number of agents. However, in real world scenarios this may not always be the case.

For example, there may be a number of robots who have a number of tasks to do. Some robots cannot perform certain tasks so robots are teamed. But there may be more tasks than robots in each team, and robots will need to prioritise. Some research has already been done in this area, but this uses *TAPF* as a basis [27].

Recursive M*

In [5], an extension to *M** is described called recursive *M**. It could increase the performance of *M** on regular *MAPF* problems. Whether recursive *M** makes a difference combined with matching, has not yet been verified.

11. Reproducibility

Results in this paper have been generated using an implementation of *M** made in Python specifically for this research. The code for this, together with the maps and raw experimental results, are publicly available on Github at https://github.com/jonay2000/research-project, doubly licensed under the Apache 2.0 and MIT licenses. Reports of bugs and new additions to this repository are always welcome.

References

[1] J. Mulderij, B. Huisman, D. Tönissen, K. van der Linden, and M. de Weerdt, "Train unit shunting and servicing: A real-life application of multi-agent path finding," 2020.

[2] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. Kumar, and S. Koenig, "Lifelong multi-agent path finding in large-scale warehouses," 2020.

[3] A. Mahdavi and M. Carvalho, "Distributed coordination of autonomous guided vehicles in multi-agent systems with shared resources," in *2019 SoutheastCon*, IEEE, 2019, pp. 1–7.

[4] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, "On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman's problem"," *The International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, 1984.

[5] G. Wagner and H. Choset, "M*: A complete multirobot path planning algorithm with performance bounds," in *international conference on intelligent robots and systems*, IEEE, 2011, pp. 3260–3267.

[6] T. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, 2010.

[7] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," 2016.

[8] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[9] R. Baauw, "Todo," 2021.

[10] E. Lam, P. Le Bodic, D. D. Harabor, and P. J. Stuckey, "Branch-and-cut-and-price for multi-agent pathfinding.," in *IJCAI*, 2019, pp. 1289–1296.

[11] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, and J. Schaeffer, "Enhanced partial expansion a*," *Journal of Artificial Intelligence Research*, vol. 50, pp. 141–187, 2014.

[12] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. Sturtevant, J. Schaeffer, and R. C. Holte, "Partial-expansion a* with selective node generation," *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, pp. 180–181, 2012.

[13] G. Sharon, R. Stern, M. Goldenberg, and A. Felner, "The increasing cost tree search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 195, pp. 470–495, 2013.

[14] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. Kumar, *et al.*, "Multi-agent pathfinding: Definitions, variants, and benchmarks," 2019.

[15] C. Ferner, G. Wagner, and H. Choset, "Odrm* optimal multirobot path planning in low dimensional search spaces," in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 3854–3859.

[16] I. de Bruin, "Todo," 2021.

[17] J. de Jong, "Todo," 2021.

[18] T. van der Woude, "Todo," 2021.

[19] E. Boyarski, A. Felner, R. Stern, G. Sharon, D. Tolpin, O. Betzalel, and E. Shimony, "Icbs: Improved conflict-based search algorithm for multi-agent pathfinding," in *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[20] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T. S. Kumar, and S. Koenig, "Adding heuristics to conflict-based search for multi-agent path finding," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, 2018.

[21] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Meta-agent conflict-based search for optimal multi-agent path finding.," *SoCS*, vol. 1, pp. 39–40, 2012.

[22] J. Li, A. Felner, E. Boyarski, H. Ma, and S. Koenig, "Improved heuristics for multi-agent path finding with conflict-based search.," in *IJCAI*, 2019, pp. 442–449.

[23] N. Jadoenathmisier, "Extending cbs to efficiently solve mapfw," 2020.

[24] A. Michels, "Multi-agent pathfinding with waypoints using branch-price-and-cut," 2020.

[25] S. Siekman, "Extending a* to solve multi-agent pathfinding problems with waypoints," 2020.

[26] J. van Dijk, "Solving the multi-agent path finding with waypoints problem using subdimensional expansion," 2020.

[27] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh, "Generalized target assignment and path finding using answer set programming," in *Twelfth Annual Symposium on Combinatorial Search*, 2019.